Putting All Together (Projet)

TP6 – DSLs, Xtext, compilers, variants*

Le but du projet a été/est de développer un langage dédié à un domaine (DSL) pour un domaine donné (CSV, JSON, classification ML ou régression ML). Le DSL doit comprendre au moins deux compilateurs et un interpréteur : à la fin, nous aurons au moins trois variantes pour exécuter un programme écrit dans votre DSL.

Ce que nous attendons, c'est qu'à la fin de votre projet, vous puissiez :

- Spécifier la syntaxe abstraite d'un DSL (avec un métamodèle)
- Spécifier la syntaxe concrète d'un DSL (avec une grammaire)
- Développer des transformations de modèles pour construire différents outils pour un DSL
- Maîtriser les bibliothèques existantes pour exécuter automatiquement des programmes et donner du sens à un DSL
- Développer des compilateurs capables de produire différents programmes exécutables à partir d'une spécification textuelle

Il y a eu plusieurs étapes/TPs pour parvenir à ce résultat :

- conception de méta-modèle pour votre DSL avec ou sans Xtext (TP1/TP2)
- conception de la syntaxe concrète de votre DSL avec Xtext (TP1/TP2)
- le "parsing", le premier compilateur, puis le deuxième (TP3/TP4)
- test de votre DSL pour notamment vérifier le bon fonctionnement de vos compilateurs (TP5) ou bien trouver des bogues fonctionnels et non fonctionnels dans vos compilateurs/interprètes ou bien encore dans... les bibliothèques/outils existants (cf BONUS)

Nous nous concentrerons sur le dernier point: tester votre DSL est un moyen de démontrer comment l'utiliser mais aussi de s'assurer de certaines qualités de votre DSL (notamment la grammaire/le métamodèle, et les compilateurs/interprètes).

L'objectif de cette dernière étape (et de ce dernier TP) est de **comparer les trois variantes (les deux compilateurs et l'interpréteur)**, à la fois en termes d'aspect fonctionnel (c'est-à-dire que le code généré doit donner le même[1] résultat lorsqu'il est exécuté, quel que soit le compilateur ou l'interpréteur) et d'aspects non fonctionnels (par exemple, le temps d'exécution).

[1] Il peut y avoir quelques exceptions : dans ce cas, veuillez en justifier la raison. Le terme "même" a des significations différentes et est volontairement vague : il peut s'agir d'une relation de stricte égalité ou d'équivalence, selon le domaine d'intérêt. Le domaine du "machine learning" ne permettra pas de se reposer sur une stricte égalité par exemple: à vous de définir une relation entre les résultats, typiquement avec un seuil de tolérance sur les écarts entre variantes.

Etant donné un *benchmark* (un ensemble de programmes écrits dans votre DSL), nous cherchons à répondre à plusieurs questions:

- Quelle est la "meilleure" variante ?
- Y a-t-il des bugs fonctionnels dans certaines variantes ?
    - Une approche immédiate consiste à contrôler si les variantes donnent le même résultat (nous l'espérons !). Dans le cadre de votre évaluation, vous pouvez utiliser des programmes spécifiés manuellement ou même des programmes générés de manière aléatoire
    - S'il existe une différence fonctionnelle, est-elle due à vos compilateurs/interprètes ? Ou est-elle due à un bug dans les bibliothèques existantes qui sont utilisées dans le cadre du code généré ?
- Y a-t-il des écarts de performance de certaines variantes ?
    - Des écarts importants peuvent suggérer des bogues liés aux performances de certaines variantes. Il peut également s'agir d'un comportement normal : la bibliothèque ciblée n'est peut-être tout simplement pas aussi efficace pour la tâche spécifiée dans votre programme.
    - Pour le CSV et le JSON, nous recommandons de se concentrer sur le "temps d'exécution" ou la "consommation de mémoire". Pour ML, nous recommandons de se concentrer sur les métriques ML (par exemple la précision)
- De nombreux programmes peuvent être utilisés, mais seuls quelques-uns entraînent des variations de performances. Ne pouvons-nous pas utiliser un sous-ensemble de programmes uniquement pour réduire le temps nécessaire à l'évaluation des variantes ?

Il est fortement recommandé de calculer des données tabulaires de ce genre lors de la mesure de performances:

Benchmark	Variant	Result?	Time	Accuracy
p1.ml	Python scikit-learn	0.86	2.36	0.86
p1.ml	R	0.92	2.34	0.92
p2.ml	Python scikit-learn	0.56	2.35	0.56
p2.ml	R	0.76	1.15	0.76
p3.ml	Python scikit-learn	0.77	0.35	0.77
p3.ml	R	0.9	1.56	0.9
			...	
			...	

Ou de ce genre:

Benchmark	Variant	Result?	Time	Memory
p1.mcsv	Python	value1	2.36	0.86
p1.mcsv	R	value1	2.34	0.92
p2.mcsv	Python	valueZZZ	2.35	0.56
p2.mcsv	R	valueYYY	1.15	0.76
p3.mcsv	Python	19	0.35	0.77
p3.mcsv	R	19	1.56	0.9
			...	
			...	

Les outils statistiques/d'analyse des données peuvent ensuite être exploités pour calculer et recueillir des informations intéressantes (typiquement via des scripts Python ou R).

Quelques autres recommandations :

- **Automatisez tout** : évitez tout effort manuel ! Vous pouvez notamment automatiser l'analyse des données (fichier CSV) et vérifier l'équivalence fonctionnelle des variantes sur de nombreux programmes. Vos résultats doivent être reproductibles
- Développer des **Docker/Dockerfile** pour héberger les outils spécifiques requis (ou au moins pour documenter les outils/bibliothèques nécessaires pour exécuter les programmes de votre DSL), ce qui permettra d'exécuter vos programmes compilés dans un environnement dédié
- Comme nous nous intéressons au temps d'exécution, il convient de faire quelques mises en garde :
    - Répéter plusieurs fois les mesures (et reporter la moyenne/l'écart type)
    - Garder à l'esprit que de nombreux facteurs peuvent influencer les mesures (par exemple, le matériel, la charge de travail de la machine, la version de la JVM)
- Votre DSL peut fournir beaucoup de constructions et il peut être nécessaire de faire un certain effort pour couvrir toutes les constructions possibles de votre DSL... vous pouvez être pragmatique et ne cibler qu'un sous-ensemble de votre DSL lors de la compilation/interprétation. Bien entendu, ce sous-ensemble ne doit pas être trop petit et, dans tous les cas, il convient de documenter explicitement quel sous-ensemble est réellement pris en charge (avec des cas de test pertinents).

Comment rendre votre travail ?

Il y aura deux rendus: un par binôme, un individuel.

Le premier rendu, par binôme, comptera pour 50% de la note globale. Il s'agit de pousser dans votre git repo. les projets Xtext, le code, le test, les programmes, les Docker file, etc ainsi qu'un README.md pour présenter le projet et comment l'utiliser.

Le deuxième rendu, individuel, consiste à rédiger un rapport (30% de la note globale). Ce rapport devra répondre aux quatre questions ci-dessus. Vos réponses doivent être étayées par des données (visualisations, statistiques, tableaux, etc.). Vous devez également inclure :

- une description technique de vos expériences (quelles variantes vous utilisez, quels programmes, les conditions de mesure, etc.)
- des explications sur comment reproduire vos expériences: un correcteur vérifiera scrupuleusement que les instructions et les artefacts (eg Docker) permettent d'effectivement reproduire les données
- indiquez les données sur lesquelles vous vous êtes appuyés.

Envoyer votre rapport au format PDF mathieu.acher@irisa.fr

Deadline: 20 janvier 2021 pour les deux rendus
Deadline: 12 février 2021 pour les deux rendus (MIAGE)

BONUS

BONUS #1 : si vous trouvez des bugs dans les bibliothèques existantes (grâce à votre DSL)
BONUS #2 : si vous parvenez à interopérer avec 2 sous-projets complémentaires
- Montrer le cas de test, si ok bonus pour les 3 équipes.